CS 403, Assignment 2

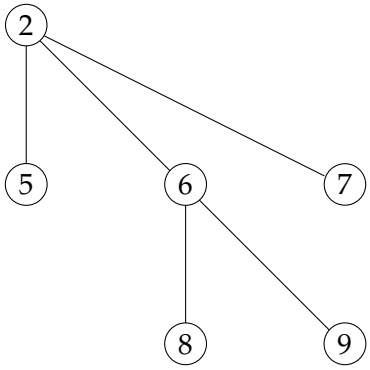
Due on 7 November at 11:59 pm

In this assignment we will manipulate trees (binary or otherwise).

For any type `a`, a binary tree over `a` is either empty or consists of a value of type `a` and exactly two sub-binary trees over `a` (either or both of which might be empty). Recall that we defined binary trees in HASKELL as follows:

```
data BinTree a = Empty | Node a (BinTree a) (BinTree a)
```

1. For any type `a`, a tree over a consists of a root value of type `a` and a (finite, possibly empty) sequence of sub-trees over `a`. For example, the following is a pictorial representation of a tree over `Integer`; it has root value 2 and three sub-trees, the first and third of which have no sub-trees, and the second of which has two sub-trees:



   A forest is then a set of trees.

   Define the HASKELL types `Tree` and `Forest` such that, for any type `a`, `Tree a` and `Forest a` are suitable for representing trees and forests over `a`, respectively.

2. One of the reasons binary trees are important is that there is a natural one-to-one correspondence between forests and binary trees, where a forest is a list of trees: A forest node `f` can be represented as a binary tree node with the first child of `f` as left sub-tree and the right sibling of `f` as the right sub-tree. Therefore one can define HASKELL functions:

```
toBinTree  ::  Forest a -> BinTree a
 toForest  ::  BinTree a -> Forest a
```

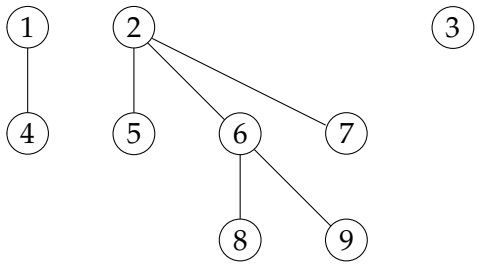
   that are mutual inverses, so that

```
toForest (toBinTree f) == f
```

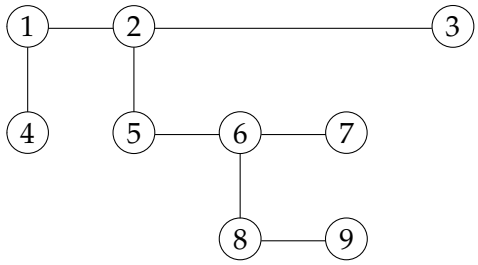
for any forest `f`, and

```
toBinTree (toForest b) == b
```

for any binary tree `b`. For example, the following forest over `Integer`:



is uniquely represented by the following binary tree:



where the first sub-binary tree of a node is drawn *below* the node, the second sub-binary tree is drawn to the *right* of the node, and empty binary trees are not shown. Define the functions `toBinTree` and `toForest`.

3. Given a tree `t`, a *trace* in `t` is the sequence of nodes on some paths in the tree that is either empty or starts with the root node of `t` and ends with some (not necessarily leaf) node; let `traces t` be the set of all the traces in `t`. For example exactly all the traces of the tree shown in Question 1 (expressed as HASKELL lists) are:

   ```
   [[], [2], [2,5], [2,6], [2,7], [2,6,8], [2,6,9]]
   ```

   You are asked to obtain the traces of a tree (binary or otherwise) using a single function `traces` for both types. For this purpose implement a type class `Traceable` that expresses the property of some HASKELL type to have traces, and then make both trees and binary trees instances of `Traceable`.

4. Make sure that your trees (both general and binary) are instances of `Show`, `Eq`, and `Ord`. The `Ord` and `Eq` instantiation should implement a custom ordering, as follows: For some trees `t` and `u` we have

   $$\texttt{t} \leq \texttt{u} \quad \text{iff} \quad \texttt{traces t} \subseteq \texttt{traces u}$$
   $$\texttt{t} == \texttt{u} \quad \text{iff} \quad \texttt{traces t} \subseteq \texttt{traces u} \text{ and } \texttt{traces u} \subseteq \texttt{traces t}$$

   Pay attention to the fact that you are thus defining a partial order over trees (that is, two trees may not be comparable with each other).

Implementation note

If you have a type with a parameter (such the type `Tree a` for trees holding values of type `a`) it is often the case that the "big" type (`Tree a`) can be an instance of some class (say, `Eq`) only if the "inner" type (`a`) is an instance of a certain type class (say, `Ord`). This kind of constraints must be specified at instantiation time. In the example mentioned above the instance `Tree a` of `Eq` will thus go as follows:

```
instance Ord a => Eq (Tree a) where
    ...
```

Note that I used `Eq` and `Ord` merely as examples, I am not claiming that the particular example above makes actual sense.

Submission guidelines

Submit a single plain text file that can be loaded in the HASKELL interpreter. It would be nice if you can submit a literate script (`.lhs` extension), but this is not required. Your script will only be loaded in the interpreter and will not go anywhere near the compiler, so do not provide anything having to deal with the HASKELL compiler such as a `main` function or I/O operations. Also provide (as suitable comments) session listings or equivalent that demonstrate and test your functions and classes. Include at the top of your script a comment with the names and emails of all the collaborators.

Submit your script by email. Recall that assignments can be solved in groups (of maximum three students), and a single solution per group should be submitted. Also recall that a penalty of 10% per day will be applied to late submissions.